Interpreted and Compiled

Sometimes it is useful to classify according to how the program is treated to make a runnable object, as this affects (a) the code development and (b) how the code is delivered

Interpreted and Compiled

Sometimes it is useful to classify according to how the program is treated to make a runnable object, as this affects (a) the code development and (b) how the code is delivered

Compiled to native machine code and executed directly: C, C++, Fortran, . . .

Interpreted and Compiled

Sometimes it is useful to classify according to how the program is treated to make a runnable object, as this affects (a) the code development and (b) how the code is delivered

Compiled to native machine code and executed directly: C, C++, Fortran, . . .

Bytecode: compile to a machine-independent code that is then interpreted or further compiled to machine code to execute. Java, Python, C#, Perl, Lua, Forth, Clisp . . .

Interpreted and Compiled

Sometimes it is useful to classify according to how the program is treated to make a runnable object, as this affects (a) the code development and (b) how the code is delivered

Compiled to native machine code and executed directly: C, C++, Fortran, . . .

Bytecode: compile to a machine-independent code that is then interpreted or further compiled to machine code to execute. Java, Python, C#, Perl, Lua, Forth, Clisp . . .

Interpreted: Basic, HTML, . . .

Interpreted and Compiled

Feet

- C#: You forget precisely how to use the .NET interface and shoot yourself in the foot. You sue Microsoft for damages

Interpreted and Compiled

Feet

- C#: You forget precisely how to use the .NET interface and shoot yourself in the foot. You sue Microsoft for damages
- C# (2): You copy how Java shot itself in the foot. Then you explain to everybody who will listen how you did it better

Interpreted and Compiled

Feet

- C#: You forget precisely how to use the .NET interface and shoot yourself in the foot. You sue Microsoft for damages
- C# (2): You copy how Java shot itself in the foot. Then you explain to everybody who will listen how you did it better
- C# (3): You can create and shoot a gun in C#, but you can't shoot your foot in managed code

Interpreted and Compiled

Feet

- Lua: You come up with a decent way to shoot yourself in the foot, but you're unsure if it's the optimal way to go about it. You ask the mailing list. Someone points out that Lua has a "shoot foot" function built in, but it's only exposed via the C API. The discussion devolves into a long debate about whether various functions should be exposed, how objects and OOP should be implemented, and whether nil should be a valid table index

Interpreted and Compiled

Feet

- Lua: You come up with a decent way to shoot yourself in the foot, but you're unsure if it's the optimal way to go about it. You ask the mailing list. Someone points out that Lua has a "shoot foot" function built in, but it's only exposed via the C API. The discussion devolves into a long debate about whether various functions should be exposed, how objects and OOP should be implemented, and whether nil should be a valid table index
- Lua (2): You shoot yourself in the foot while watching enviously how Scheme shoots you in the foot

Interpreted and Compiled

Compiling to native machine code for a specific processor produces fast running programs, using all the facilities of the hardware (when done properly)

Interpreted and Compiled

Compiling to native machine code for a specific processor produces fast running programs, using all the facilities of the hardware (when done properly)

Provides lots of error checking in the compilation phase

Interpreted and Compiled

Compiling to native machine code for a specific processor produces fast running programs, using all the facilities of the hardware (when done properly)

Provides lots of error checking in the compilation phase

Uses the compile-run-edit cycle of development

Interpreted and Compiled

Compiling to native machine code for a specific processor produces fast running programs, using all the facilities of the hardware (when done properly)

Provides lots of error checking in the compilation phase

Uses the compile-run-edit cycle of development

Some programmers regard this cycle as “slow” as they keep having to wait for the compiler to do its thing, probably spitting out errors they have to fix. So this is less preferred for rapid prototyping of code

Interpreted and Compiled

But even compile-run-edit can support moderately fast development

Interpreted and Compiled

But even compile-run-edit can support moderately fast development

Most systems support modules (or equivalent) that can be separately compiled, giving

(a) faster compilation times
(b) error checking on small units of code

Interpreted and Compiled

But even compile-run-edit can support moderately fast development

Most systems support modules (or equivalent) that can be separately compiled, giving

(a) faster compilation times
(b) error checking on small units of code

Only when all the modules compile would we need to link them into a working executable to test

Interpreted and Compiled

And the programmer has to produce a fairly complete outline of their code that passes the compiler checks before they can do a test run

Interpreted and Compiled

And the programmer has to produce a fairly complete outline of their code that passes the compiler checks before they can do a test run

Compared to interpreting, where bits of code can be incomplete, or even function definitions missing and it can still run (a bit)

Interpreted and Compiled

And the programmer has to produce a fairly complete outline of their code that passes the compiler checks before they can do a test run

Compared to interpreting, where bits of code can be incomplete, or even function definitions missing and it can still run (a bit)

For coders whose testing process is “try it and see”, a compiler is a hindrance

Interpreted and Compiled

And the programmer has to produce a fairly complete outline of their code that passes the compiler checks before they can do a test run

Compared to interpreting, where bits of code can be incomplete, or even function definitions missing and it can still run (a bit)

For coders whose testing process is “try it and see”, a compiler is a hindrance

Other people are happy to wait for a compiler as it does a lot of checking and produces fast running code

Interpreted and Compiled

But also:

Interpreted and Compiled

But also:

An interpreter typically finds just one error per run

Interpreted and Compiled

But also:

An interpreter typically finds just one error per run

And maybe takes a long time to hit an error; if at all

Interpreted and Compiled

But also:

An interpreter typically finds just one error per run

And maybe takes a long time to hit an error; if at all

Compilers can find several errors per compile

Interpreted and Compiled

But also:

An interpreter typically finds just one error per run

And maybe takes a long time to hit an error; if at all

Compilers can find several errors per compile

So the arguments for rapid development time are not at all clear

Interpreted and Compiled

> *"Rapid development" is just another way of saying "Getting it wrong quickly"*

Anon

> *. . . also called the "run-crash-modify" code cycle*

Anon

Interpreted and Compiled

Exercise Consider this C code:

```
#include <stdio.h>

int main(void)
{
  int sum = 0;
  for (int i = 0; i < 100; i++) {
    sum += i;
  }

  printf("%d\n", sum);
  return 0;
}
```

Compile using optimisation (`-O2`) and look at the machine code it produces. Then remove the `printf` line and repeat. Do the same with other languages of your choice.

Interpreted and Compiled

Other remarks on interpreted languages:

Interpreted and Compiled

Other remarks on interpreted languages:

You have portable, compact code — the source

Interpreted and Compiled

Other remarks on interpreted languages:

You have portable, compact code — the source

But you need an interpreter for each architecture you want to run on

Interpreted and Compiled

Other remarks on interpreted languages:

You have portable, compact code — the source

But you need an interpreter for each architecture you want to run on

And the interpreter occupies memory space itself

Interpreted and Compiled

Other remarks on interpreted languages:

You have portable, compact code — the source

But you need an interpreter for each architecture you want to run on

And the interpreter occupies memory space itself

So not really that portable or compact

Interpreted and Compiled

Other remarks on interpreted languages:

You have portable, compact code — the source

But you need an interpreter for each architecture you want to run on

And the interpreter occupies memory space itself

So not really that portable or compact

Secrecy of code is poor: you can't stop people reading your intellectual property

Interpreted and Compiled

Other remarks on interpreted languages:

You have portable, compact code — the source

But you need an interpreter for each architecture you want to run on

And the interpreter occupies memory space itself

So not really that portable or compact

Secrecy of code is poor: you can't stop people reading your intellectual property

Exercise Are there any major interpreted languages these days?

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

A kind of machine code for a standardised, *virtual machine*

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

A kind of machine code for a standardised, *virtual machine*

A real machine can then use a runtime system to interpret the bytecode and execute the program

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

A kind of machine code for a standardised, *virtual machine*

A real machine can then use a runtime system to interpret the bytecode and execute the program

Or further compile the bytecode to native machine code and run that

Interpreted and Compiled

In contrast, some languages compile to a machine independent *bytecode*

A kind of machine code for a standardised, *virtual machine*

A real machine can then use a runtime system to interpret the bytecode and execute the program

Or further compile the bytecode to native machine code and run that

Or a mixture of the two!

Interpreted and Compiled

Bytecode produces more compact compiled code: a single bytecode instruction might correspond to a large number of machine code instructions and so is sometimes suitable for small memory machines

Interpreted and Compiled

Bytecode produces more compact compiled code: a single bytecode instruction might correspond to a large number of machine code instructions and so is sometimes suitable for small memory machines

For example, Forth is used in embedded controllers

Interpreted and Compiled

Bytecode produces more compact compiled code: a single bytecode instruction might correspond to a large number of machine code instructions and so is sometimes suitable for small memory machines

For example, Forth is used in embedded controllers

This is good as long as the runtime that executes the bytecode is not too large

Interpreted and Compiled

Bytecode produces more compact compiled code: a single bytecode instruction might correspond to a large number of machine code instructions and so is sometimes suitable for small memory machines

For example, Forth is used in embedded controllers

This is good as long as the runtime that executes the bytecode is not too large

For example, Java has a large and complex runtime

Interpreted and Compiled

The bytecode is machine independent so allowing mobile code, working across different architectures and different operating systems

Interpreted and Compiled

The bytecode is machine independent so allowing mobile code, working across different architectures and different operating systems

“Compile once and run anywhere”

Interpreted and Compiled

The bytecode is machine independent so allowing mobile code, working across different architectures and different operating systems

“Compile once and run anywhere”

Even mobile in the sense the program can move between different processors while it is running

Interpreted and Compiled

A bytecode compiler is a compiler, so it can provide lots of error checking in the compilation phase, so tends to be “slow” development

Interpreted and Compiled

A bytecode compiler is a compiler, so it can provide lots of error checking in the compilation phase, so tends to be “slow” development

Generally a modest overhead in loss of speed in the execution of the bytecode, though claims are often made that a good VM and compiler could eliminate this overhead

Interpreted and Compiled

A bytecode compiler is a compiler, so it can provide lots of error checking in the compilation phase, so tends to be “slow” development

Generally a modest overhead in loss of speed in the execution of the bytecode, though claims are often made that a good VM and compiler could eliminate this overhead

Note that Go (native compiler) and Python (bytecode) have deliberately fast compilers (omitting some analysis and optimisation) to mitigate the perceived compiler overhead

Interpreted and Compiled

A bytecode compiler can't take advantage of specifics of the hardware it runs on, though the VM *might* be able to

Interpreted and Compiled

A bytecode compiler can't take advantage of specifics of the hardware it runs on, though the VM *might* be able to

The VM *can* do a lot of optimisations

Interpreted and Compiled

A bytecode compiler can't take advantage of specifics of the hardware it runs on, though the VM *might* be able to

The VM *can* do a lot of optimisations

For example, Java compilers generally don't do a lot of optimisation, they regard optimisation as the job of the VM

Interpreted and Compiled

A bytecode compiler can't take advantage of specifics of the hardware it runs on, though the VM *might* be able to

The VM *can* do a lot of optimisations

For example, Java compilers generally don't do a lot of optimisation, they regard optimisation as the job of the VM

So Java VMs tend to be huge, complicated things

Interpreted and Compiled

A bytecode compiler can't take advantage of specifics of the hardware it runs on, though the VM *might* be able to

The VM *can* do a lot of optimisations

For example, Java compilers generally don't do a lot of optimisation, they regard optimisation as the job of the VM

So Java VMs tend to be huge, complicated things

These days, a typical Java compiler starts by interpreting the bytecode and then compiles the heavily-used sections of code as it runs your program

Interpreted and Compiled

A bytecode compiler can't take advantage of specifics of the hardware it runs on, though the VM *might* be able to

The VM *can* do a lot of optimisations

For example, Java compilers generally don't do a lot of optimisation, they regard optimisation as the job of the VM

So Java VMs tend to be huge, complicated things

These days, a typical Java compiler starts by interpreting the bytecode and then compiles the heavily-used sections of code as it runs your program

Exercise Read about JIT compilation and its advantages and disadvantages

Interpreted and Compiled

The platform independence of a VM is not actually used much in real life

Interpreted and Compiled

The platform independence of a VM is not actually used much in real life

Maybe Java is not as platform independent as you might think; or it's not worth the testing, or the marketing to target more than one architecture?

Interpreted and Compiled

Exercise Read about the bytecodes for Java, Python, C#, Perl (Parrot), Pascal (P-code), Forth, Lua, Clisp and others and discover how they are generated and how they are executed

Exercise WASM, the new-ish way of executing code in a Web browser, is a bytecode. It is designed for streaming: it can be further compiled (and even executed) while still being downloaded. Read about it

Exercise There are more architectures supported by C/C++ than are supported by a Java VM. Discuss

Interpreted and Compiled

Note: any given language can be compiled/interpreted/run in any of these ways

Interpreted and Compiled

Note: any given language can be compiled/interpreted/run in any of these ways

Though languages do tend to have a preferred approach

Interpreted and Compiled

Note: any given language can be compiled/interpreted/run in any of these ways

Though languages do tend to have a preferred approach

For example, C is almost always compiled, Java is bytecoded, while Basic tends to be interpreted

Interpreted and Compiled

Note: any given language can be compiled/interpreted/run in any of these ways

Though languages do tend to have a preferred approach

For example, C is almost always compiled, Java is bytecoded, while Basic tends to be interpreted

But there are C interpreters and Java to machine code compilers

Interpreted and Compiled

Exercise Look at several languages and determine their usual methods of execution

Exercise Then determine the positives and negatives of doing it differently (e.g., compiling Java to machine code; bytecoding C)

Interpreted and Compiled

Where Java has been called “compile once, run anywhere”, C is sometimes called “compile anywhere, run anywhere”

Interpreted and Compiled

Where Java has been called “compile once, run anywhere”, C is sometimes called “compile anywhere, run anywhere”

C can be thought as a “high level assembly language” that is mostly machine independent

Interpreted and Compiled

Where Java has been called “compile once, run anywhere”, C is sometimes called “compile anywhere, run anywhere”

C can be thought as a “high level assembly language” that is mostly machine independent

And is often the first language to be supported on new architectures, so is widely available

Interpreted and Compiled

Where Java has been called “compile once, run anywhere”, C is sometimes called “compile anywhere, run anywhere”

C can be thought as a “high level assembly language” that is mostly machine independent

And is often the first language to be supported on new architectures, so is widely available

It’s usually the first supported language as operating systems are often written in C

Interpreted and Compiled

Where Java has been called “compile once, run anywhere”, C is sometimes called “compile anywhere, run anywhere”

C can be thought as a “high level assembly language” that is mostly machine independent

And is often the first language to be supported on new architectures, so is widely available

It’s usually the first supported language as operating systems are often written in C

> *Java — Write once, problems everywhere*

Anon

Interpreted and Compiled

Other compilers leverage existing frameworks

Interpreted and Compiled

Other compilers leverage existing frameworks

TypeScript and *CoffeeScript* are both “more sophisticated” languages that compile to JavaScript

Interpreted and Compiled

Other compilers leverage existing frameworks

TypeScript and *CoffeeScript* are both "more sophisticated" languages that compile to JavaScript

Note carefully: the output from the compiler is JavaScript, which is then executed in the normal way

Interpreted and Compiled

Other compilers leverage existing frameworks

TypeScript and *CoffeeScript* are both "more sophisticated" languages that compile to JavaScript

Note carefully: the output from the compiler is JavaScript, which is then executed in the normal way

This is *source-to-source compilation* and such compilers are called *transpilers*

Interpreted and Compiled

"Transpiling" turns out to be another vague concept when you look at it carefully

Interpreted and Compiled

"Transpiling" turns out to be another vague concept when you look at it carefully

Even though, say, the `clang` compiler converts C to LLVM, the *Low Level Virtual Machine* language, this is not generally regarded as transpiling, as LLVM is a kind of machine-independent assembly language

Interpreted and Compiled

"Transpiling" turns out to be another vague concept when you look at it carefully

Even though, say, the `clang` compiler converts C to LLVM, the *Low Level Virtual Machine* language, this is not generally regarded as transpiling, as LLVM is a kind of machine-independent assembly language

Perhaps better to say: transpilation is source to source compilation *at the same level of abstraction*, typically a high level language to a high level language

Interpreted and Compiled

"Transpiling" turns out to be another vague concept when you look at it carefully

Even though, say, the `clang` compiler converts C to LLVM, the *Low Level Virtual Machine* language, this is not generally regarded as transpiling, as LLVM is a kind of machine-independent assembly language

Perhaps better to say: transpilation is source to source compilation *at the same level of abstraction*, typically a high level language to a high level language

Assembly language to assembly language transpilers also exist

Interpreted and Compiled

This approach allows the language designer to create a new or “better” language without needing a lot of the bothersome parts of a full compiler that take a lot of work to implement and optimise

Interpreted and Compiled

This approach allows the language designer to create a new or "better" language without needing a lot of the bothersome parts of a full compiler that take a lot of work to implement and optimise

For example, TypeScript has static compile time type checking and classes

Interpreted and Compiled

This approach allows the language designer to create a new or “better” language without needing a lot of the bothersome parts of a full compiler that take a lot of work to implement and optimise

For example, TypeScript has static compile time type checking and classes

Thus allowing programmers to write code in a higher level and better structured way that still runs in a browser

Interpreted and Compiled

This approach allows the language designer to create a new or "better" language without needing a lot of the bothersome parts of a full compiler that take a lot of work to implement and optimise

For example, TypeScript has static compile time type checking and classes

Thus allowing programmers to write code in a higher level and better structured way that still runs in a browser

And allows TypeScript to use all the hard work the JavaScript implementors have done in making JavaScript run quickly

Interpreted and Compiled

A transpiler is a compiler, so it can do the usual things a compiler might do

Interpreted and Compiled

A transpiler is a compiler, so it can do the usual things a compiler might do

A transpiler might

Interpreted and Compiled

A transpiler is a compiler, so it can do the usual things a compiler might do

A transpiler might

- do the usual syntax and other checking of the source language

Interpreted and Compiled

A transpiler is a compiler, so it can do the usual things a compiler might do

A transpiler might

- do the usual syntax and other checking of the source language
- do type checking — so the target could have weaker type system

Interpreted and Compiled

A transpiler is a compiler, so it can do the usual things a compiler might do

A transpiler might

- do the usual syntax and other checking of the source language
- do type checking — so the target could have weaker type system
- do optimisations — less likely, as that's something you want from the target language compiler

Interpreted and Compiled

Note the source and target languages can be very different!

Interpreted and Compiled

Note the source and target languages can be very different!

An accidental benefit is that you get easy integration of your source language with the with target language and its libraries

Interpreted and Compiled

For example, consider this Typescript code:

```
// typed variables
function add(a: number, b: number): number {
    return a + b;
}

// classes
class Thing {
    private value: number;
    constructor(n: number) {
        this.value = n;
    }
    toString(): string {
        return `${this.value}`;
    }
}
```

Interpreted and Compiled

This might compile to JavaScript

```
// typed variables
function add(a, b) {
    return a + b;
}
// classes
var Thing = /** @class */ (function () {
    function Thing(n) {
        this.value = n;
    }
    Thing.prototype.toString = function () {
        return "" + this.value;
    };
    return Thing;
}());
```

Interpreted and Compiled

The Typescript transpiler can do type checking, while a JavaScript compiler can't

Interpreted and Compiled

The Typescript transpiler can do type checking, while a JavaScript compiler can't

But has had the benefit of being typechecked in the transpilation process

Interpreted and Compiled

The Typescript transpiler can do type checking, while a JavaScript compiler can't

But has had the benefit of being typechecked in the transpilation process

Exercise The Typescript code `function add(a: number, b: number)` has type information that could be used to generate code that is better optimised than the transpiled JavaScript `function add(a, b)`. Think about this

Interpreted and Compiled

Typescript is quite close to JavaScript, so the output code from the transpiler is fairly simple and human-readable

Interpreted and Compiled

Typescript is quite close to JavaScript, so the output code from the transpiler is fairly simple and human-readable

Exercise Transpiling languages that are far apart can produce very complex code. Experiment with transpiling Python to C, or Lisp to C, or Haskell to C, or whatever

Interpreted and Compiled

Typescript is quite close to JavaScript, so the output code from the transpiler is fairly simple and human-readable

Exercise Transpiling languages that are far apart can produce very complex code. Experiment with transpiling Python to C, or Lisp to C, or Haskell to C, or whatever

Exercise Conway's Life is Turing Complete: you can build a "computer" out of gliders etc. There is even a compiler from C to Life. Read about this

Interpreted and Compiled

C is a popular target language, as C compilers are widely available and have been heavily optimised

Interpreted and Compiled

C is a popular target language, as C compilers are widely available and have been heavily optimised

Exercise Read about Kyoto Common Lisp that compiles to C

Exercise Read about Cambridge Common Lisp that compiles to C (speed), mixed with bytecode (compact)

Exercise The original C++ compiler was a transpiler to C. Read about this

Interpreted and Compiled

Similarly, given the widespread availability of Java, the Java virtual machine is a popular target

Interpreted and Compiled

Similarly, given the widespread availability of Java, the Java virtual machine is a popular target

It is debatable whether we should call these transpilers, but they are exploiting existing infrastructure in a similar way

Interpreted and Compiled

Similarly, given the widespread availability of Java, the Java virtual machine is a popular target

It is debatable whether we should call these transpilers, but they are exploiting existing infrastructure in a similar way

Exercise Read about Clojure, a Lisp that compiles to Java bytecode (and JavaScript and .NET)

Exercise Read about Scala, a functional language that compiles to Java bytecode (and JavaScript and native code)

Interpreted and Compiled

Exercise Read about Kotlin, a statically and implicitly typed OO language that compiles to Java bytecode (and JavaScript and native code). Currently one of Google's preferred languages for Android app development

Exercise Read about Groovy, a scripting language that compiles to Java bytecode

Exercise Read about GraalVM that compiles Python, JavaScript, Ruby, R, LLVM and more to the Java VM

Interpreted and Compiled

And JavaScript; these have the advantage the result can run in a browser

Interpreted and Compiled

And JavaScript; these have the advantage the result can run in a browser

Exercise Read about Dart, a C# style language that compiles to JavaScript

Exercise Read about Elm, a functional language that compiles to JavaScript

Exercise Read about Nim, a implicitly typed “multi-paradigm” language that compiles to JavaScript (and C, C++, Objective C)

Interpreted and Compiled

A closely related topic is: does this language run natively, or does it need a runtime?

Interpreted and Compiled

A closely related topic is: does this language run natively, or does it need a runtime?

For example, C compiles all the way down to native machine code and a C program requires little else to run: access to the OS kernel and systems libraries, but nothing more than that

Interpreted and Compiled

A closely related topic is: does this language run natively, or does it need a runtime?

For example, C compiles all the way down to native machine code and a C program requires little else to run: access to the OS kernel and systems libraries, but nothing more than that

Sometimes this is described as “running on the bare metal”

Interpreted and Compiled

A closely related topic is: does this language run natively, or does it need a runtime?

For example, C compiles all the way down to native machine code and a C program requires little else to run: access to the OS kernel and systems libraries, but nothing more than that

Sometimes this is described as “running on the bare metal”

While Java compiles to a bytecode that need to be itself interpreted or further compiled

Interpreted and Compiled

So Java requires a *runtime* infrastructure

Interpreted and Compiled

So Java requires a *runtime* infrastructure

A program to run your program

Interpreted and Compiled

So Java requires a *runtime* infrastructure

A program to run your program

You don't directly run your bytecompiled Java program, but you actually are running the `java` runtime program which loads and executes your program

Interpreted and Compiled

So Java requires a *runtime* infrastructure

A program to run your program

You don't directly run your bytecompiled Java program, but you actually are running the `java` runtime program which loads and executes your program

This runtime infrastructure includes other support the language needs, like a garbage collector

Interpreted and Compiled

So Java requires a *runtime* infrastructure

A program to run your program

You don't directly run your bytecompiled Java program, but you actually are running the `java` runtime program which loads and executes your program

This runtime infrastructure includes other support the language needs, like a garbage collector

Exercise There were some experiments to build hardware to execute Java bytecode. Read about this

Interpreted and Compiled

Similarly, Python needs a VM to execute its bytecode

Interpreted and Compiled

Similarly, Python needs a VM to execute its bytecode

Go and Erlang need support for their thread mechanisms; and so on

Interpreted and Compiled

Similarly, Python needs a VM to execute its bytecode

Go and Erlang need support for their thread mechanisms; and so on

Native machine code is good for low-level applications, like operating systems, or on embedded systems, where runtime languages would be a cumbersome overhead

Interpreted and Compiled

Similarly, Python needs a VM to execute its bytecode

Go and Erlang need support for their thread mechanisms; and so on

Native machine code is good for low-level applications, like operating systems, or on embedded systems, where runtime languages would be a cumbersome overhead

On the other hand, runtimes provide useful programming features (like GC and lightweight threads)

Interpreted and Compiled

So, again, your problem to solve should lead you to pick the right kind of approach

Interpreted and Compiled

So, again, your problem to solve should lead you to pick the right kind of approach

Must the program have as few dependencies as possible or use as few resources as possible?

Interpreted and Compiled

So, again, your problem to solve should lead you to pick the right kind of approach

Must the program have as few dependencies as possible or use as few resources as possible?

Or are we less concerned with size and brute performance and are happy to employ a large, many featured infrastructure?

Interpreted and Compiled

Exercise Many people think C is a simple language. Have a look at `https://www.nayuki.io/page/summary-of-c-cpp-integer-rules`

Exercise Then look at how C (and similar other languages) treat *undefined behaviour*

Exercise And *Implementation defined behaviour*

Bits and Pieces

Now for a brief run-though of some assorted smaller bits and pieces

Bits and Pieces

Now for a brief run-though of some assorted smaller bits and pieces

Covering some features that you might like to consider when choosing a programming language

Bits and Pieces

Compile/Run or Read-Eval-Print Loop

Bits and Pieces

Compile/Run or Read-Eval-Print Loop

REP: the systems reads a line, evaluates it, then prints the result

Bits and Pieces

Compile/Run or Read-Eval-Print Loop

REP: the systems reads a line, evaluates it, then prints the result

Interactive, good for beginners to get an immediate feel for what is happening and good for quick code hacking

Bits and Pieces

Compile/Run or Read-Eval-Print Loop

REP: the systems reads a line, evaluates it, then prints the result

Interactive, good for beginners to get an immediate feel for what is happening and good for quick code hacking

Examples: Python, Haskell, Lisp

Bits and Pieces

Compile and Run: good for catching errors, good for optimising code, good for large bodies of code

Bits and Pieces

Compile and Run: good for catching errors, good for optimising code, good for large bodies of code

Examples: C, Java, C++, Python, Haskell, Lisp

Expression Based or Statement Based

Expression Based or Statement Based

Statement based: the code is a sequence of statements to be executed

Expression Based or Statement Based

Statement based: the code is a sequence of statements to be executed

Statements can contain expressions to be evaluated

Expression Based or Statement Based

Statement based: the code is a sequence of statements to be executed

Statements can contain expressions to be evaluated

```
if (x > 10) {
  y = x + 2;
}
else {
  y = x + 3;
}
```

Expression Based or Statement Based

Statement based: the code is a sequence of statements to be executed

Statements can contain expressions to be evaluated

```
if (x > 10) {
  y = x + 2;
}
else {
  y = x + 3;
}
```

Examples: C, Java, C++

Bits and Pieces

Expressions/Statements

Expression based: the code is a sequence of expressions to be evaluated

Bits and Pieces

Expressions/Statements

Expression based: the code is a sequence of expressions to be evaluated

```
y = x + if (x > 10) { 2 } else { 3 };
```

Bits and Pieces

Expressions/Statements

Expression based: the code is a sequence of expressions to be evaluated

```
y = x + if (x > 10) { 2 } else { 3 };
```

Examples: Lisp, Rust

Expression based: the code is a sequence of expressions to be evaluated

```
y = x + if (x > 10) { 2 } else { 3 };
```

Examples: Lisp, Rust

The value can be ignored if you don't need it, making it effectively a statement

Bits and Pieces

Expressions/Statements

In C we have statements

```
int inc(int n) {
  if (n > 0) {
    return n + 1;
  }
  else {
    return n - 1;
  }
}
```

`if` is a statement

While Rust has expressions

```
fn inc(n: i32) -> i32 {
  if n > 0 {
    n + 1
  }
  else {
    n - 1
  }
}
```

Bits and Pieces

Expressions/Statements

While Rust has expressions

```
fn inc(n: i32) -> i32 {
  if n > 0 {
    n + 1
  }
  else {
    n - 1
  }
}
```

`if` is an expression that returns a value

The function body is an expression that returns a value

Bits and Pieces

Expressions/Statements

Expression-based is more flexible, more general, and possibly less wordy than statement-based, but some people claim it is a bit harder to read

Bits and Pieces

Expressions/Statements

Expression-based is more flexible, more general, and possibly less wordy than statement-based, but some people claim it is a bit harder to read

Though others claim that expressions are easier to read

Bits and Pieces

Expressions/Statements

Expression-based is more flexible, more general, and possibly less wordy than statement-based, but some people claim it is a bit harder to read

Though others claim that expressions are easier to read

Exercise What would you expect the assignment expression "`x = 42`" to return?

Exercise Read about how `;` might be a *statement terminator* or a *expression separator*

Note that C has `y = x + (x>10) ? 2 : 3;`

and Python has `y = x + 2 if (x>10) else 3`

for the `if` case, but are otherwise mostly statement languages

Multiple value return

Functions that return more than one value:

```
fn sumdiff(a, b) {
  return a+b, a-b;
}
x, y = sumdiff(4,5);
```

Examples: Lisp, Maple, Go

Bits and Pieces

Multiple values

Exercise What about Python:

```
def sumdiff(a, b):
  return a+b, a-b
```

Exercise Some languages support pairs:

```
fn sumdiff(a: i32, b: i32) -> (i32, i32) {
  (a+b, a-b)
}
```

What is the difference between returning two values and returning one value, which is a pair?